Context Preserving Crossover in Genetic Programming.

Patrik D'haeseleer
LSI Logic
197 Ravenswood Ave.
Atherton, CA, 94027
pdhaes@lsil.com
pdhaes@cs.stanford.edu

Abstract:

This paper introduces two new crossover operators for Genetic Programming (GP). Contrary to the regular GP crossover, the operators presented attempt to preserve the context in which subtrees appeared in the parent trees. A simple coordinate scheme for nodes in an S-expression tree is proposed, and crossovers are only allowed between nodes with exactly or partially matching coordinates.

1 Introduction:

One of the main differences between Genetic Programming (GP) and classical Genetic Algorithm (GA) implementations is the fact that crossover in GP does not preserve any kind of context in the chromosome. This is because the standard crossover as defined by Koza [Koza 1992] swaps subtrees which are chosen at random in both parent trees.

The "cut and splice" crossover used in classical bit string GA's leads to lexical convergence, because crossing two identical individuals will yield the same individuals again. This lexical convergence tends to lead to loss of diversity in the population however, which is why GA's often need to use a certain amount of mutation to keep the evolution going. As Koza points out in [Koza 1992], random subtree crossover maintains diversity because crossing two identical trees can yield completely different trees. This is at the expense of lexical convergence however. Although the individuals in a GP run seem to converge functionally, they never seem to converge lexically.

Using random subtree crossover in GP, useful subtrees, or "Building Blocks", tend to spread out over the entire tree. Andrew Singleton and Nick Keenan [Singleton 1993] interpret this phenomenon as "defense against crossover": *'[...] Expressions grow because they are defending themselves against the destructive effects of crossover, and attempting through redundancy to preserve their current fitness in the next generation.'* This proliferation of Building Blocks throughout the entire tree usually has a positive effect on fitness but tends to make the trees excessively large and eventually still leads to a de facto convergence. *'In the later stages of the population's evolution, [...] the mean fitness approaches a quasi-stationary state. [...] Code is then in competition for the robustness of its behavior in the face of the actions of the genetic operators.'* [Altenberg 1994]

This lack of lexical convergence may be harmful for certain classes of complex problems in which the solutions tend to require a certain amount of structure. For example in the "Central Place Food Foraging" ant colony problem [Koza 1992], the internal structure of an individual typically consists of a decision tree built from functions (such as IF-FOOD-HERE, IF_CARRYING_FOOD etc.), while the leaf nodes of the tree consist of actions (like PICK-UP, MOVE_RANDOM etc.) to be performed in the context defined by their place in the decision tree. In problems such as these, crossing over two subtrees from different parts of a tree may have harmful effects, because these two subtrees will most likely come from different contexts in their parent trees.

One of the key factors in the success of Koza style Automatically Defined Functions (ADF's) [Koza 1992, Koza 1994] is precisely that the crossover used there is highly constrained: crossover only takes place between matching branches. This effectively separates the evolution of each branch, allowing them to specialize without disruptive influence of code from a different branch. ADF's are fairly artificial though, in that they require the user to specify the number of ADF's to use for a certain problem, and function and terminal sets for each ADF and for the main program tree.

Two new crossovers are introduced that attempt to preserve the context in which subtrees appear by restricting the crossover to take place only between subtrees in similar locations (defined in terms of their *node coordinates* in their respective parent trees). This gives us a more flexible and implicit mechanism to uncouple the evolution of the different branches of the individual.

2 Node Coordinates

The position of each node in a rooted tree can be uniquely identified by specifying the path to be followed from the root to reach this specific node. A node's position can therefore be described by a tuple of n coordinates T = (b_1, b_2, ..., b_n), where n is the depth of the node in the tree, and b_i indicates which branch to choose at level i (counting left to right). For example, (2,1,3,1) would be the node found by taking the 2nd branch at the root of the tree, then take the first branch at that level, then the third branch there, and then the first.

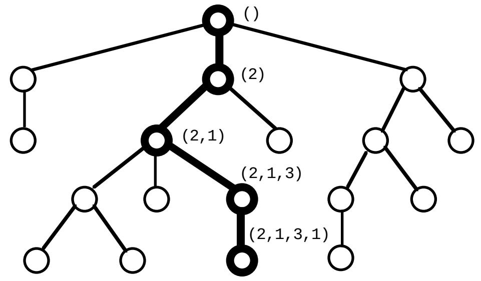

Figure 1: Rooted tree with tree coordinates.

3 Strong Context Preserving Crossover (SCPC):

SCPC only allows subtrees to be chosen for crossover if the nodes at which these subtrees are found in the parent trees have *exactly* the same node coordinates (see Figure 2).

This operator imposes quite severe restrictions on which nodes of the tree are eligible to be selected for crossover. In fact, it can easily be seen that using this technique, a subtree generated at a certain level in a certain part of a tree can never migrate to another level, nor to any other section of the tree.

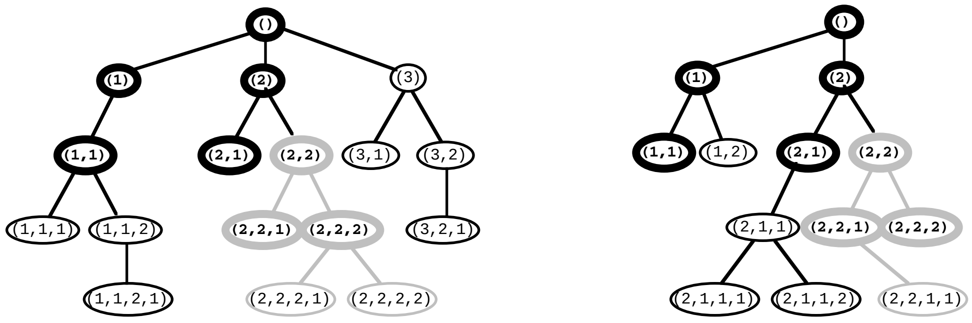

Figure 2: Two parent trees selected for crossover. Nodes with matching coordinates (can be selected as crossover points) are indicated in bold. Gray segments in both parents show a possible choice of subtrees to be exchanged.

This would cause major problems in terms of diversity if we were to use SCPC exclusively, because to be able to cover the entire search space every possible function and terminal would have to appear at every possible node coordinate at least once throughout the entire population. As in regular GA's, this problem can be somewhat alleviated by adding a certain amount of mutation to regularly reintroduce functions and terminals at new positions.

A more important problem with using this operator is that it *totally* prohibits good building blocks from spreading to other parts of the tree. For most problems, this evolutionary isolation of subtrees is overkill. Although it perfectly protects subtrees from irrelevant code coming from other branches, it also makes the solutions more brittle with respect to crossover, because good building blocks can no longer build up any redundancy. Also, if a certain Building Block can be of use in different regions of the tree, it can no longer migrate from one to the other, but will have to evolve in each region independently. The solution to this is to use a certain mix of SCPC, and the regular crossover. This also eliminates the need for extra mutation.

4 Weak Context Preserving Crossover (WCPC):

As mentioned above, SCPC usually restricts the choice off crossover subtrees too much, such that an additional amount of regular crossover has to be added to make the system work at all. Another way of achieving the same effect is to relax the requirement that the node coordinates of the subtrees have to match exactly.

I have implemented WCPC as a simple variant of SCPC. The subtree to be exchanged in the first parent is chosen from the set of nodes for which there is a matching node in the second parent, as in SCPC. The subtree in the second parent however, is a random subtree of the node that matches the first subtree. In other words, if T_1 and T_2 are a valid choice of subtrees for crossover with SCPC (i.e. the nodes at T_1 and T_2 have the same node coordinates in their respective parent trees), then T_1 and any random subtree $T_2' \subseteq T_2$ (T_2 inclusive) is a valid choice of subtrees for crossover with WCPC.

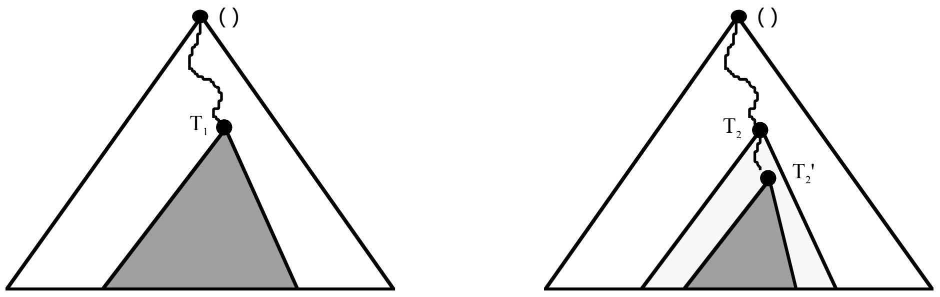

Figure 3: Schematic representation of subtree selection for crossover with WCPC. If T_1 and T_2 are valid subtrees for SCPC (i.e. the nodes at T_1 and T_2 have the same coordinates), then T_1 and $T_2' \subseteq T_2$ are valid subtrees for WCPC.

Note that although this crossover is asymmetric with respect to the parent trees, this has no noticeable effect on the population since both parents are chosen in the same manner and are therefore basically interchangeable.

5 Results:

Both crossover operators were tested on a series of four different problems from [Koza 1992] and [Koza 1994]. The four problems were chosen to give a wide range in expected usefulness of Context Preserving Crossover.

For the runs using SCPC and regular crossover, a mix of 100/0, 75/25, 50/50, 25/75, and 0/100 was used. For the runs using WCPC, no regular crossover was added. All runs were run under approximately the same conditions as in [Koza 1992] and [Koza 1994], with exception of an additional 10% mutation.

The results of 10 (for 5.1 and 5.2) or 15 runs (5.3 and 5.4) with different initial random seed were collected in order to calculate the "Effort" E (minimum number of individuals that need to be processed) required to reach a solution with 99% certainty. This is based on the cumulative probability of success P(M,i) (solid curve in figures 4-7), and I(M,i,z) (gray curve in figures 4-6). See [Koza 1992, Koza 1994] for details on how to calculate P(M,i), I(M,i,z), E and other performance metrics.

5.1 Obstacle Avoiding Robot

This problem consists of evolving a program for a mobile robot that mops a square (toroidal) grid of floor tiles within a given maximum number of operations. Two fitness cases are used, each with a different placement of six one-square obstacles. (For further details, see [Koza 1994])

Because the robot only gets to execute its program one single time, the solution will have to be a very elaborate tree, containing all moves it needs to make to mop the entire floor. This problem is almost made to be solved using ADF's, because what you need here is a couple of small intelligent subroutines (such as "move forward, and look for an obstacle"), which are spread out over the entire tree. In fact, a bunch of calls to ADF's, strung together by PROGN's would suffice to solve this problem. Because of this, we would expect SCPC and WCPC to perform fairly badly, because they actually keep blocks of code from spreading out throughout the tree.

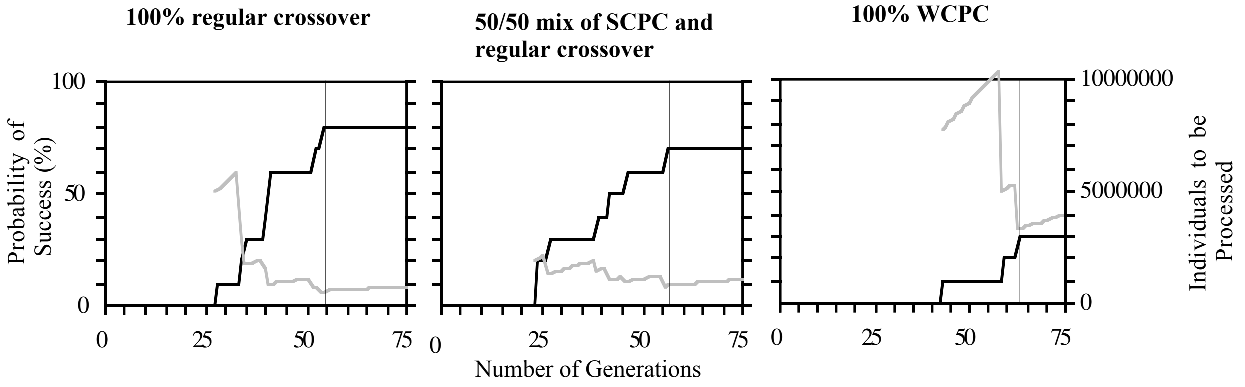

Figure 4: Performance curves for Obstacle Avoiding Robot. Individuals to be processed (I(M,i,z)) is indicated in gray, cumulative probability of success (P(M,i)) in black. Vertical bar is number of generations at which I (M,i,z) is minimal.

The results of the runs confirm this: using regular crossover, E = 660,000 individuals need to be processed to be 99% certain to achieve a successful result . The best result using SCPC is for the 50/50 mix: E = 912,000. Although most of the runs with this operator mix actually finish earlier than for the regular crossover, the performance is slightly worse because only 7/10 runs finish, as opposed to 8/10 for regular crossover. WCPC performs even worse, with only 3/10 runs finished after 75 generations, it needs E = 3,328,000 individuals to be processed.

5.2 Boolean 11-Multiplexer

The goal of this problem is to find a function which, given 8 data bits and 3 address bits, returns the contents of the data bit whose address is given by the three address bits. Function set used is F = {AND, OR, NOT, IF}. (For further details, see [Koza 1992])

Number of individuals needed to be processed for regular crossover is 300,000. Best mix for SCPC is again at 50/50: E = 132,000. A significant improvement over using just regular crossover! 100% WCPC does again fairly badly: E = 512,000.

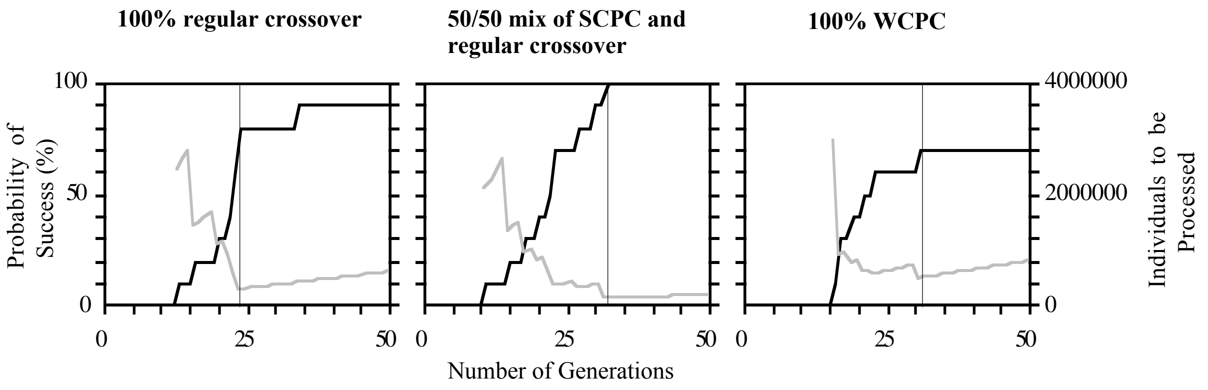

Figure 5: Performance curves for Boolean 11-Multiplexer.

5.3 Obstacle Avoiding Robot - Iterated Version

The original description of the Obstacle Avoiding Robot in [Koza 1994] calls for one single execution of the evolved program. Given the fact that only two different fitness cases (i.e. different obstacle configurations) are used, it is quite likely that the resulting solution has been overfit to these fitness cases, effectively optimizing the program tree to solve those two floor layouts. A more realistic version of this problem would be to iteratively evaluate the program until all floor tiles are mopped or a given maximum number of operations have been performed.

Whereas the goal in 5.1 was to make a very bushy program tree, with many small building blocks spread throughout the tree, this version of the problem implies the construction of a reactive plan that behaves intelligently, no matter on which tile the robot starts. It is clear that being able to preserve context under crossover is potentially very useful in evolving such a plan.

Note that this problem is considerably easier than the non-iterated version, although its solution is likely to contain more "intelligence" and to be more general. In fact, solutions for the non-iterated version form a (very small) subset of the solutions for the iterated version. There is a whole range of possible solutions, going from doing one single intelligent operation per iteration (real reactive plan), to doing all operations in one single iteration (solution for non-iterated version). The larger the plan executed per iteration, the more information about the floor plan will have to be stored in the program itself.

Using regular crossover, we need to process 152,000 individuals for a 99% probability of success (about four times less than for the non-iterated version). The best mix for SCPC is this time at 75% SCPC, 25% regular crossover: E = 76,000. Again, we see a significant improvement in processing required to solve this problem using SCPC. Unlike the two previous problems, this problem also has a very high success rate using pure SCPC: 13/15 runs finished within 50 generations (none of the runs using 100% SCPC for 5.1 and 5.2 finished).
The performance of WCPC is again disappointing: E = 276,000.

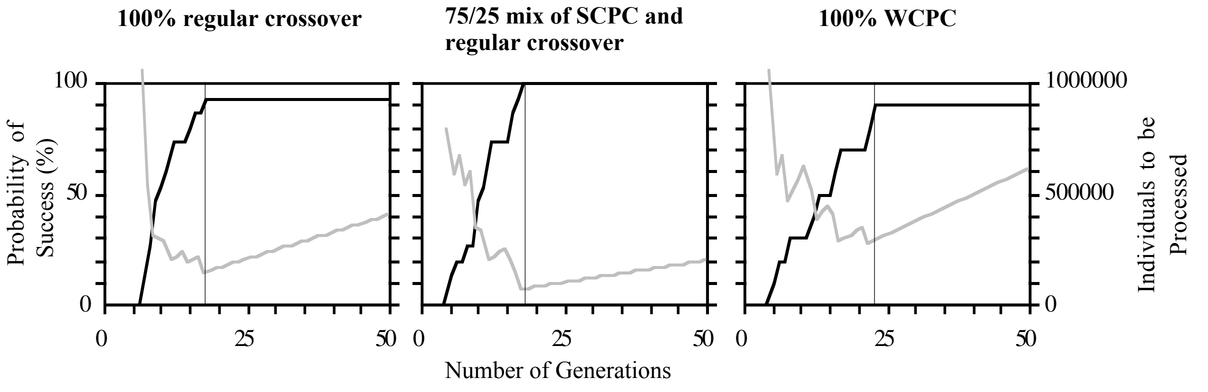

Figure 6: Performance curves for Iterated Version of Obstacle Avoiding Robot.

5.4 Central Place Food Foraging

The Central Place Food Foraging problem consists of breeding a common computer program that, when simultaneously executed by all "ants" in an ant colony, causes the transportation of the available food to the nest of the colony. (For further details, see [Koza 1992])

Contrary to 5.3, it is not possible to find an efficient solution for this problem that only requires one single iteration of the program (for each ant). This is because efficient food gathering requires constant interaction and cooperation between the ants (by means of pheromone trails which lead the other ants to the food). Therefore, the programs evolved will tend to be fairly short (after elimination of all useless code), containing only a couple of operations per iteration. This implies that context preserving crossover could be potentially very useful.

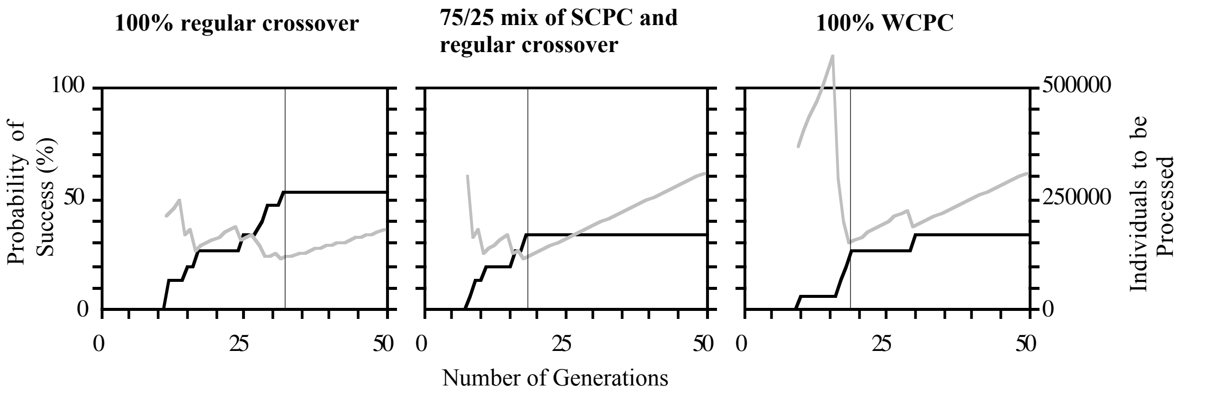

Figure 7: Performance curves for Central Place Food Gathering problem.

Again, a mix of 75% SCPC and 25% regular crossover (E = 114,000) outperforms 100 % regular crossover (E = 115,500). The difference is insignificant in view of the small number of runs (15) used to compute these values.

However, the solutions generated by using only the regular crossover tend to be much larger than the ones generated by SCPC (on average about twice as large for this problem). This seems to support the observation by Andrew Singleton [Singleton 1993] and others that random subtree crossover makes the trees large and complex.

Memory requirements aside, larger trees to evaluate also means that the processing time needed per individual increases. However, the trees generated by regular crossover seem to contain a larger percentage of "dead branches" which will never get executed, such that the execution time per individual is not as long as one would expect from the size of the trees being executed. Still, total processing time needed to reach a solution using SCPC turns out to be noticeably smaller than with regular crossover alone.

For WCPC, E is 150,000. Again, processing time per individual is shorter for WCPC due to smaller trees generated, but the difference in number of individuals to be processed is too large to be able to outperform regular crossover.

6 Conclusions

A mix of Strong Context Preserving Crossover with regular crossover gives in most cases superior performance over pure regular crossover. The worse performance for the Obstacle Avoiding Robot is most likely due to the fact that for that problem it is desirable to spread out good Building Blocks throughout the tree as much as possible. As mentioned earlier, the iterated version of the Obstacle Avoiding Robot (for which SCPC performs a lot better) is a much more realistic way to implement this problem, leading to a smaller program and a more general solution.

Both new operators result in smaller individuals to be evaluated because they do not tend to spread out Building Blocks over the entire tree, as the regular crossover does. Whether this results in a faster evaluation of individuals depends on what proportion of the code really get executed. In any case, it reduces the memory requirements.

100% Weak Context Preserving Crossover performs fairly badly on all of the problems studied here. It is not clear what causes this discrepancy in performance between WCPC and SCPC + regular crossover. Possibly the fact that the resistance for subtrees to migrate "horizontally" (i.e. from branch to branch) is far greater when using only WCPC, compared to using a mix with regular crossover. On the other hand, 100% WCPC yielded successful results for all problems, whereas 100% SCPC only did so for the last two. Apparently, relaxing the restriction on crossover does have some positive result.

7 Future work

- Clearly, the effect of Context Preserving Crossover on the structure of the resulting trees and the propagation of Building Blocks need to be investigated in more detail, as well as the effects on tree size and execution speed.

- The two operators presented (and the tree coordinate system in general) have a serious flaw. Looking back at Figure 2, it is clear that none of the nodes in the third branch of the first parent is eligible for crossover, because the second parent only has two branches at the root node. In general, if there are functions with different numbers of arguments (and therefore nodes with different numbers of branches), then the rightmost branches at each level in the tree will be less likely to be selected for crossover, resulting in different evolution speeds. It should be feasible to modify the tree coordinate system to eliminate this bias. For instance by using fractions of the form b_i/B_i as coordinates, where b_i indicates which branch to choose at level i and B_i stands for the total number of branches at that level.

- The crossover used with ADF's can be described in terms of a partial matching of tree coordinates as well. The first coordinate of each node specifies in which branch of the root the node is located, and therefore in which ADF (or main program) the node occurs. Crossover is then restricted to subtrees which have the same first tree coordinate. It is clear that a whole variety of other partial matching schemes based on tree coordinates could be tried.

Acknowledgments

Thanks to Dr. John Koza and James Rice for their help with the problem implementations, and to all the people on genetic-programming@cs.stanford.edu for the seminal ideas.

All the runs in this paper were done with a modified version of Walter Alden Tackett's Simple Genetic Programming in C (SGPC) system. Many thanks to LSI Logic for the use of their Sparc-10's, and to the other members of LSI's Advanced Development Lab for letting me hoard CPU cycles at night.

Bibliography

Altenberg, L. (1994). The Evolution of Evolvability in Genetic Programming. In K. E. Kinnear, editor, *Advances in Genetic Programming*, Chapter 3, Cambridge, MA. MIT. Press.

Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA. MIT Press.

Koza, J. (1994). *Genetic Programming II: Automatic Discovery of Reusable Subprograms.* Cambridge, MA. MIT Press. To appear.

Singleton, A. (1993). Greediness. Posting of 8-21-1993 on *genetic-programming@cs.stanford.edu* mailing list.